Instance Attributes in Python

Introduction

An object is useful because it can hold its own data.

For example, if we create three students:

```
student1 = Student("Hafsa", 9)
student2 = Student("Asim", 10)
student3 = Student("Ali", 9)
```

Each student needs to remember different information.

That information is stored using **instance attributes**.

An instance attribute is a piece of data that belongs to a **specific object**.

The key idea is:

```
Class
  
Creates objects
  
Each object gets its own instance attributes
```

For example:

```
student1
    name = "Hafsa"
    grade = 9
```

```
student2
□□□ name = "Asim"
□□□ grade = 10
```

The two objects come from the same class, but their instance data is different.

---

1. What Is an Instance Attribute?

An instance attribute is an attribute stored on an individual object.

Example:

```
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Create an object:

```
student = Student("Hafsa", 25)
```

Here:

```
student.name
student.age
```

are instance attributes.

They belong specifically to `student`.

---

2. Why Are They Called "Instance" Attributes?

An object is an **instance** of a class.

For example:

```
class Student:
    pass
```

Then:

```
student1 = Student()
student2 = Student()
```

Both are instances of `Student`.

If we give them data:

```
student1.name = "Hafsa"
student2.name = "Asim"
```

`name` is an instance attribute because each instance can have a different value.

```
Student
 □
 □□□ student1
 □     □□□ name = Hafsa
 □
 □□□ student2
       □□□ name = Asim
```

---

3. Creating Instance Attributes With `__init__()`

The most common way to create instance attributes is inside `__init__()`.

```
class User:

    def __init__(self, username, email):
        self.username = username
        self.email = email
```

Create an object:

```
user = User("hafsa", "hafsa@example.com")
```

Now the object has:

```
username
email
```

We can access them:

```
print(user.username)
print(user.email)
```

Output:

```
hafsa
hafsa@example.com
```

The attributes belong to that particular `user` object.

---

4. Understanding `self`

Instance attributes are usually written using `self`.

```
self.name
self.age
self.email
self.balance
```

`self` refers to the current object.

Example:

```
class Student:

    def __init__(self, name):
        self.name = name
```

When we create:

```
student1 = Student("Hafsa")
```

`self` refers to `student1`.

So:

```
self.name = name
```

becomes conceptually:

```
student1.name = "Hafsa"
```

When we create:

```
student2 = Student("Asim")
```

`self` refers to `student2`.

So:

```
self.name = name
```

becomes:

```
student2.name = "Asim"
```

This is why each object gets its own value.

---

5. Instance Attribute vs Parameter

Consider:

```
class Student:

    def __init__(self, name):
        self.name = name
```

There are two `name`s here, but they have different roles.

```
name
 
Parameter
```

`self.name`

`Instance attribute`

When we write:

```
student = Student("Hafsa")
```

the value flows like this:

`"Hafsa"`

`name parameter`

`self.name`

`student.name`

So:

```
print(student.name)
```

returns:

```
Hafsa
```

---

6. Every Object Can Have Different Values

This is one of the most important properties of instance attributes.

```
class Student:
```

```
    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Create objects:

```
student1 = Student("Hafsa", 9)
student2 = Student("Asim", 10)
```

Now:

```
print(student1.name)
print(student2.name)
```

Output:

```
Hafsa
Asim
```

And:

```
print(student1.grade)
print(student2.grade)
```

Output:

```
9
10
```

The class defines the structure, but each instance stores its own values.

---

7. Changing an Instance Attribute

Instance attributes can normally be changed after the object has been created.

```
class Student:

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Create:

```
student = Student("Hafsa", 9)
```

Change the grade:

```
student.grade = 10
```

Now:

```
print(student.grade)
```

Output:

```
10
```

The object has been updated.

---

8. Changing One Object Does Not Change Another

Consider:

```
class Student:

    def __init__(self, name, grade):
        self.name = name
        self.grade = grade
```

Create:

```
student1 = Student("Hafsa", 9)
student2 = Student("Asim", 10)
```

Change:

```
student1.grade = 10
```

Now:

```
print(student1.grade)
print(student2.grade)
```

Output:

```
10
10
```

Both happen to be `10`, but for different reasons.

Originally:

```
student1.grade → 9
student2.grade → 10
```

After the change:

```
student1.grade  10
student2.grade  10
```

Changing `student1` did not modify `student2`.

---

9. Adding an Instance Attribute Later

Python also allows attributes to be added after an object is created.

```
class User:

    def __init__(self, name):
        self.name = name
```

Create:

```
user = User("Hafsa")
```

Later:

```
user.age = 25
```

Now:

```
print(user.name)
print(user.age)
```

Output:

```
Hafsa
25
```

However, this should be used carefully.

If every `User` is supposed to have an `age`, it is usually better to define it inside `__init__()`:

```
class User:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

This makes the object's expected structure clear.

---

10. Different Objects Can Have Different Attributes

Python objects don't necessarily have to contain exactly the same attributes.

For example:

```
class User:

    def __init__(self, name):
        self.name = name
```

Create:

```
user1 = User("Hafsa")
user2 = User("Asim")
```

Add an attribute to only one:

```
user1.age = 25
```

Now:

```
user1
□□□ name
□□□ age

user2
□□□ name
```

This is possible in Python.

But in well-designed application code, it is generally better to keep the object's expected attributes consistent.

---

11. Checking Whether an Attribute Exists

You can use `hasattr()`:

```
class User:

    def __init__(self, name):
        self.name = name

user = User("Hafsa")

print(hasattr(user, "name"))
print(hasattr(user, "age"))
```

Output:

```
True
False
```

`hasattr()` asks:

> Does this object currently have this attribute?

---

12. Accessing Attributes With `getattr()`

Python provides `getattr()` for dynamic attribute access.

```
class User:

    def __init__(self, name):
        self.name = name

user = User("Hafsa")

print(getattr(user, "name"))
```

Output:

```
Hafsa
```

You can also provide a default value:

```
print(getattr(user, "age", 0))
```

Output:

```
0
```

This is useful when the attribute may not exist.

---

13. Setting Attributes With `setattr()`

`setattr()` allows you to create or change an attribute dynamically.

```
class User:
    pass

user = User()

setattr(user, "name", "Hafsa")
```

Now:

```
print(user.name)
```

Output:

```
Hafsa
```

This is equivalent to:

```
user.name = "Hafsa"
```

The normal dot notation is easier to read, while `setattr()` becomes useful when the attribute name is determined dynamically.

---

14. Removing an Instance Attribute

You can remove an attribute using `del`.

```
class User:

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Create:

```
user = User("Hafsa", 25)
```

Delete:

```
del user.age
```

Now:

```
print(user.name)
```

still works.

But:

```
print(user.age)
```

raises:

```
AttributeError
```

because the attribute no longer exists.

---

15. Instance Attributes Can Store Any Data Type

An instance attribute can contain:

String

```
self.name = "Hafsa"
```

Integer

```
self.age = 25
```

Boolean

```
self.is_active = True
```

List

```
self.skills = ["Python", "JavaScript"]
```

Dictionary

```
self.settings = {
    "theme": "dark",
    "notifications": True
}
```

Another object

```
self.profile = Profile(...)
```

This makes objects flexible enough to represent complex real-world entities.

---

16. Instance Attributes With Lists

A very common pattern is storing a list inside each object.

```
class Student:

    def __init__(self, name):
        self.name = name
        self.subjects = []
```

Create two students:

```
student1 = Student("Hafsa")
student2 = Student("Asim")
```

Add a subject:

```
student1.subjects.append("Python")
```

Check:

```
print(student1.subjects)
print(student2.subjects)
```

Output:

```
['Python']
[]
```

Each student has a separate list.

---

17. Why the List Should Be Created Inside `__init__()`

Consider:

```
class Student:

    def __init__(self, name):
        self.name = name
        self.subjects = []
```

Every time `__init__()` runs, a new list is created.

```
student1 → subjects → []
student2 → subjects → []
student3 → subjects → []
```

These lists are independent.

This is different from accidentally sharing one mutable list across objects.

A safe mental model is:

```
__init__()
   ↓
new object
   ↓
new instance attributes
   ↓
new mutable containers
```

---

18. Instance Attributes and Methods

Methods can read and modify instance attributes.

```
class BankAccount:

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
```

Create:

```
account = BankAccount("Hafsa", 5000)
```

Call:

```
account.deposit(1000)
```

Now:

```
print(account.balance)
```

Output:

```
6000
```

The method changes the instance attribute:

```
self.balance
```

belonging to that specific account.

---

19. Instance State

The collection of values stored in an object is often called its **state**.

For example:

```
class User:

    def __init__(self, name, is_active):
        self.name = name
        self.is_active = is_active
```

This object:

```
user = User("Hafsa", True)
```

has state:

```
name      = Hafsa
is_active  = True
```

If we change:

```
user.is_active = False
```

the object's state changes.

This gives us an important concept:

```
Instance Attributes
      
```

State
□
What the object currently remembers

---

20. Instance Attributes vs Class Attributes

This distinction is essential.

Instance attribute

```
class Student:

    def __init__(self, name):
        self.name = name
```

Each object gets its own `name`.

Class attribute

```
class Student:

    school = "haas.dev Academy"
```

`school` belongs to the class and can be shared by instances.

Together:

```
class Student:

    school = "haas.dev Academy"

    def __init__(self, name):
```

```
        self.name = name
```

Here:

`school`
→
Class attribute

`name`
→
Instance attribute

---

21. A Useful Comparison

Instance Attribute	Class Attribute
Belongs to an instance	Belongs to the class
Usually created with `self`	Usually defined directly in the class body
Can differ between objects	Often shared
Example: `self.name`	Example: `school`
Represents object-specific state	Represents class-level data

Example:

```
class Student:

    school = "haas.dev Academy"

    def __init__(self, name):
        self.name = name
```

For:

```
student1 = Student("Hafsa")
student2 = Student("Asim")
```

we can think:

```
student1
□□□ name = Hafsa

student2
□□□ name = Asim

Student
□□□ school = haas.dev Academy
```

---

22. Attribute Lookup

When you write:

```
student.name
```

Python looks for `name` on the object.

If it doesn't find it there, Python can continue looking at the class and its inheritance hierarchy.

For example:

```
class Student:

    school = "haas.dev Academy"

    def __init__(self, name):
        self.name = name
```

Then:

```
student = Student("Hafsa")
```

Python finds:

```
student.name
```

on the instance.

For:

```
student.school
```

Python can find `school` on the class.

This distinction becomes important when working with class attributes and inheritance.

---

23. Shadowing a Class Attribute

Consider:

```
class Student:

    school = "haas.dev Academy"

student1 = Student()
student2 = Student()
```

Both can access:

```
print(student1.school)
print(student2.school)
```

Now:

```
student1.school = "Another School"
```

This creates an instance attribute named `school` for `student1`.

Now:

```
print(student1.school)
print(student2.school)
```

Output:

```
Another School
haas.dev Academy
```

The instance attribute takes precedence for `student1`.

This is called **shadowing** the class attribute.

---

24. Don't Accidentally Share Mutable State

Consider this:

```
class Team:
    members = []
```

This list belongs to the class.

So:

```
team1 = Team()
team2 = Team()

team1.members.append("Hafsa")
```

Now:

```
print(team2.members)
```

may also show:

```
['Hafsa']
```

because both objects are accessing the same class-level list.

If every team needs its own list, use:

```
class Team:

    def __init__(self):
```

```
        self.members = []
```

Now:

```
team1.members
    
separate list

team2.members
    
separate list
```

This is a very important OOP rule.

---

25. Real World Example: Shopping Cart

A shopping cart is a good example of instance state.

```
class ShoppingCart:

    def __init__(self, owner):
        self.owner = owner
        self.items = []
        self.total = 0
```

Create two carts:

```
cart1 = ShoppingCart("Hafsa")
cart2 = ShoppingCart("Asim")
```

Add an item:

```
cart1.items.append("Keyboard")
```

Now:

```
print(cart1.items)
print(cart2.items)
```

Output:

```
['Keyboard']
[]
```

Each cart maintains its own state.

---

26. Real World Example: haas.dev Resource

A resource can have object-specific information.

```
class Resource:

    def __init__(self, title, category, file_name):
        self.title = title
        self.category = category
        self.file_name = file_name
        self.views = 0
```

Create resources:

```
resource1 = Resource(
    "Python Functions",
    "Python",
```

```
    "python-functions.pdf"
)

resource2 = Resource(
    "What Is OOP?",
    "Python",
    "what-is-oop.pdf"
)
```

Each resource has its own:

```
title
category
file_name
views
```

If someone views `resource1`:

```
resource1.views += 1
```

then:

```
print(resource1.views)
print(resource2.views)
```

Output:

```
1
0
```

The state of one resource changes without changing another resource.

---

27. Designing Good Instance Attributes

Before adding an attribute, ask:

Does this value belong to one specific object?

If yes, it may be an instance attribute.

For a `User`:

```
username
email
age
```

For a `Product`:

```
name
price
stock
```

For a `BankAccount`:

```
owner
balance
```

For a `Resource`:

```
title
category
views
```

The key question is:

>

> Would two objects normally have different values for this piece of information?

If yes, it is likely object-specific state.

---

28. Don't Store Everything as an Attribute

Not every value needs to be stored.

Suppose:

```
class Rectangle:

    def __init__(self, width, height):
        self.width = width
        self.height = height
```

We could calculate the area when needed:

```
def area(self):
    return self.width * self.height
```

Instead of storing:

```
self.area = width * height
```

Why?

Because if `width` or `height` changes, a stored `area` could become outdated.

Prefer storing the fundamental state and calculating derived values when appropriate.

Stored:
width
height

Calculated:
area

This is a useful object-design principle.

---

29. Common Beginner Mistakes

Mistake 1: Forgetting `self`

Incorrect:

```
class User:

    def __init__(self, name):
        name = name
```

Correct:

```
class User:

    def __init__(self, name):
        self.name = name
```

Mistake 2: Confusing parameter and attribute

`def __init__(self, name):`

`name` is the parameter.

`self.name`

is the instance attribute.

Mistake 3: Accidentally using a class attribute

Incorrect when each object needs its own list:

```
class Cart:
    items = []
```

Better:

```
class Cart:

    def __init__(self):
        self.items = []
```

Mistake 4: Creating attributes inconsistently

This:

`user1.age = 25`

while other users don't have `age` can make code harder to reason about.

If all users require an age, define it in `__init__()`.

Mistake 5: Storing unnecessary derived values

Instead of:

```
self.total = self.price * self.quantity
```

consider calculating:

```
def total(self):
    return self.price * self.quantity
```

when the value can change and doesn't need to be stored.

---

30. Mental Model

Think of every object as its own small container.

For example:

```
student1 = Student("Hafsa", 9)
student2 = Student("Asim", 10)
```

Imagine:

```
student1
┌──────────────────────┐
│ name  = Hafsa    │
```

```
┌──────────────────┐
│ grade = 9        │
└──────────────────┘
```

```
student2
┌──────────────────┐
│ name  = Asim     │
│ grade = 10       │
└──────────────────┘
```

Same class.

Different objects.

Different instance state.

This is the core mental model.

---

31. Practical Workflow

When designing instance attributes:

Step 1: Identify the object

Example:

```
Product
```

Step 2: Identify its state

```
name
price
```

`stock`

Step 3: Decide what must exist immediately

Put required initial state in `__init__()`.

Step 4: Create the attributes

```
self.name = name
self.price = price
self.stock = stock
```

Step 5: Identify mutable state

Lists, dictionaries, and sets usually need to be created per instance.

Step 6: Add behavior

Create methods that read or modify the state.

Step 7: Keep state consistent

Avoid having some objects unexpectedly missing important attributes.

---

32. Mini Project: Task Object

Create a task manager.

Each task should have:

```
title
priority
```

```
completed
tags
```

Start with:

```
class Task:

    def __init__(self, title, priority="normal"):
        self.title = title
        self.priority = priority
        self.completed = False
        self.tags = []
```

Create tasks:

```
task1 = Task("Learn Python", "high")
task2 = Task("Build project")
```

Add tags:

```
task1.tags.append("python")
task1.tags.append("learning")
```

Complete a task:

```
task1.completed = True
```

Inspect the state:

```
print(task1.title)
print(task1.priority)
print(task1.completed)
print(task1.tags)
```

Extend the project

Add methods:

`complete()`
`add_tag()`
`remove_tag()`
`change_priority()`

The methods should modify the instance attributes of the specific task.

---

33. 5 Minute Challenge

Create a `Playlist` class.

Each playlist should have:

`name`
`songs`
`is_public`

Requirements:

- `name` comes from the user.
- `songs` starts as an empty list.
- `is_public` defaults to `False`.
- Two playlist objects must have separate song lists.

Example:

```
playlist1 = Playlist("Study")
```

`playlist2 = Playlist("Workout")`

Add a song only to `playlist1`.

Verify that `playlist2` remains empty.

---

34. Exercises

Exercise 1

Create a `Book` class with:

`title`
`author`
`pages`

Create two book objects with different values.

Exercise 2

Create a `BankAccount` class with:

`owner`
`balance`

Change the balance of one account and verify that the other account is unchanged.

Exercise 3

Create a `Student` class with:

`name`
`marks`
`subjects`

Initialize `subjects` as an empty list.

Make sure each student has a separate list.

Exercise 4

Create a `Product` class with:

`name`
`price`
`stock`

Add a method that decreases `stock` when an item is sold.

Exercise 5

Create a `Resource` class with:

`title`
`category`
`views`

Add a method:

`record_view()`

that increases the object's `views` by one.

---

35. Mini Quiz

Question 1

What is an instance attribute?

A. Data belonging to a specific object
B. Data belonging only to Python
C. A module
D. A function

Answer: A

Question 2

Which syntax commonly creates an instance attribute?

`self.name = name`

or:

`name = name`

Answer:

`self.name = name`

Question 3

What does `self` represent?

A. The class
B. The current instance
C. The parent class
D. The module

Answer: B

Question 4

If two objects have the same class, can their instance attributes contain different values?

Answer: Yes.

Question 5

Where is it usually better to create an instance's empty list?

A. As a class attribute
B. Inside `__init__()`
C. Outside Python
D. Inside a comment

Answer: B

Question 6

What happens when:

`student1.name = "Hafsa"`

is executed?

Answer:

The `name` instance attribute of `student1` is assigned `"Hafsa"`.

---

36. Interview Questions

Beginner

1. What is an instance attribute?
2. How do you create an instance attribute?
3. What does `self` mean?
4. Why are instance attributes commonly created in `__init__()`?
5. Can two objects have different values for the same instance attribute?
6. Can an instance attribute be changed after object creation?
7. Can you add an attribute after an object has been created?

Intermediate

8. What is the difference between instance and class attributes?
9. What happens when an instance attribute has the same name as a class attribute?

10. Why can a class-level mutable list cause unexpected behavior?
11. How do `hasattr()`, `getattr()`, and `setattr()` work?
12. When should a value be stored as an instance attribute versus calculated by a method?
13. What is object state?
14. Why should important instance attributes usually be initialized consistently?
15. How can instance attributes contain other objects?

---

37. Cheat Sheet

Create instance attributes

```
class User:

    def __init__(self, name, email):
        self.name = name
        self.email = email
```

Access

```
print(user.name)
```

Change

```
user.name = "Asim"
```

Add later

```
user.age = 25
```

Delete

```
del user.age
```

Check existence

```
hasattr(user, "age")
```

Dynamic access

```
getattr(user, "name")
```

Dynamic assignment

```
setattr(user, "name", "Hafsa")
```

Separate mutable state

```
class Cart:

    def __init__(self):
        self.items = []
```

Class attribute

```
class Student:
    school = "haas.dev Academy"
```

Instance attribute

```
class Student:

    def __init__(self, name):
        self.name = name
```

---

38. Key Takeaways

- An **instance attribute** belongs to a specific object.
- Instance attributes are commonly created using `self`.
- `__init__()` is the most common place to initialize them.

- Different objects can have different values for the same attribute.
- Changing one object's instance attribute normally does not change another object's attribute.
- Instance attributes represent an object's **state**.
- They can store strings, numbers, booleans, lists, dictionaries, and even other objects.
- Mutable instance data such as lists should generally be created inside `__init__()`.
- Class attributes and instance attributes are different.
- An instance attribute can shadow a class attribute with the same name.
- `hasattr()`, `getattr()`, and `setattr()` allow dynamic attribute handling.
- Store fundamental state as attributes and calculate derived values when appropriate.
- Good object design keeps important instance attributes consistent and meaningful.

The core idea is:

`CLASS`

`Creates instances`

`Each instance`

`Owns its own attributes`

`Those attributes represent its state`

Website: https://dev-roast-app.vercel.app

Visit haas.dev for more resources and guides.